Newton® Technology

JOURNAL

Volume I, Number 2 *April 1995*

Inside This Issue

Tool News

Coming Soon: Newton Toolkit 1.5 1

NewtonScript Techniques

NewtonScript Performance Tuning 1

Developer Group News

Newton Developer Evening Brings Newton Community Up to Date 3

NewtonScript Techniques

Stepping Through Routing 5

Newton News

Apple Announces New Newton MessagePad 120, Communications and Software Solutions! 11

Advanced Techniques

Advanced Debugging 20

Licensee Specifics

Marco Software Architecture, Version 1.0 22

Newton®

Tool News

Coming Soon: Newton Toolkit 1.5

by Tony Espinoza
Apple Computer, Inc.

The NTK Team learned a lot from Newton developers during the past two years. Your feedback at developer conferences and through the on-line forums has helped in our efforts to bring you a powerful, productive NTK. You can expect NTK 1.5 later this Spring. For now, here's a quick look at some of the exciting new features in 1.5.

Productivity Enhancements and New Capabilities

A primary goal for NTK has always been to provide a high-productivity environment; version 1.5 features improvements to streamline the development process. For starters, the browser is easier to navigate. The slot list now responds to single clicks instead of double clicks, and the pop-lists for *Methods* and *Attributes* have been alphabetically ordered. NTK 1.5 browsers now enable you to cut, copy, and paste views and slots without having to open a layout window.

You can also build new kinds of packages. NTK 1.0 focused on application packages. With 1.5, developers can choose from a range of different target output formats. For example, you can easily create auto, store, and dictionary parts. Perhaps the most exciting change is the new ability to convert a desktop font into a Newton font part and include it with your application.

continued on page 13

NewtonScript Techniques

© 1994, Apple Computer, Inc. and Calliope Enterprises, Inc.

NewtonScript Performance Tuning

by Julie McKeehan and Neil Rhodes
Calliope Enterprises, Inc.

There are a number of areas within the Newton system that affect Newton application performance. The more important of these areas are: NewtonScript, the view system, and Newton data storage design (soups). This article will cover the first area, NewtonScript. Optimizing the view system is discussed in "Performance and the View System," by Mike Engber in PIE Developer's Magazine.

When you want to optimize the performance of NewtonScript within your application you need to address three important areas:

- Writing generally efficient code in the first place.
- Knowing the speediest ways of doing things within NewtonScript.
- Using memory correctly; this involves knowing how to minimize your use of the frame heap memory by maximizing the amount of code you keep in ROM and having code that allocates the correct amount of memory to provide the fastest, most efficient use of RAM when you do need to use it.

Efficient Code – Optimizing the Hot Spots

Let us address each of these issues in turn, starting with some resources for writing efficient code.

continued on page 14

Newton® Technology
JOURNAL

Volume I, Number 2 *April 1995*

Published by Apple Computer, Inc.

Lee DePalma Dorsey • *Managing Editor*

Gerry Kane • *Coordinating Editor, Technical Content*

Tony Espinoza • *Coordinating Editor, Technical Tools Marketing*

David Glickman • *Coordinating Editor, Business Content*

Gabriel Acosta-Lopez • *Coordinating Editor, DTS and Training Content*

Philip Ivanier • *Manager, Newton Developer Relations*

Technical Peer Review Board
J. Christopher Bell, Bob Ebert, Jim Schram, Maurice Sharp, Steve Strong, Bruce Thompson

Contributors
David Baum, Julie McKeehan, Neil Rhodes, Patty Tulloch

Produced by Xplain Corporation

Neil Ticktin • *Publisher*

Scott T Boyd • *Editor*

John Kawakami • *Editorial Assistant*

Judith Chaplin • *Art Director*

© 1994 Apple Computer, Inc., 1 Infinite Loop, Cupertino, CA 95014, 408-996-1010. All rights reserved.

Apple, the Apple logo, APDA, AppleDesign, AppleLink, AppleShare, Apple SuperDrive, AppleTalk, HyperCard, LaserWriter, Light Bulb Logo, Mac, MacApp, Macintosh, Macintosh Quadra, MPW, Newton, Newton Toolkit, NewtonScript, Performa, QuickTime, StyleWriter and WorldScript are trademarks of Apple Computer, Inc., registered in the U.S. and other countries. AOCE, AppleScript, AppleSearch, ColorSync, develop, eWorld, Finder, OpenDoc, Power Macintosh, QuickDraw, SNA•ps, StarCore, and Sound Manager are trademarks, and ACOT is a service mark of Apple Computer, Inc. Motorola and Marco are registered trademarks of Motorola, Inc. NuBus is a trademark of Texas Instruments. PowerPC is a trademark of International Business Machines Corporation, used under license therefrom. Windows is a trademark of Microsoft Corporation and SoftWindows is a trademark used under license by Insignia from Microsoft Corporation. UNIX is a registered trademark of UNIX System Laboratories, Inc. CompuServe, Pocket Quicken by Intuit, CIS Retriever by BlackLabs, PowerForms by Sestra, Inc., ACT! by Symantec, Berlitz, and all other trademarks are the property of their respective owners.

Mention of products in this publication is for informational purposes only and constitutes neither an endorsement nor a recommendation. All product specifications and descriptions were supplied by the respective vendor or supplier. Apple assumes no responsibility with regard to the selection, performance, or use of the products listed in this publication. All understandings, agreements, or warranties take place directly between the vendors and prospective users. Limitation of liability: Apple makes no warranties with respect to the contents of products listed in this publication or of the completeness or accuracy of this publication. Apple specifically disclaims all warranties, express or implied, including, but not limited to, the implied warranties of

Editor's Note

by Lee DePalma Dorsey, Managing Editor

Newton platform news travels fast, and with the recent series of Newton announcements in the last few months, news is traveling even faster with the platform's latest entrants. The newest additions to the Newton product family include two new hardware devices – one from Motorola and one from Apple – and a number of new communications devices. With these new products come an infinite number of wireless communication solutions waiting to be built by developers and delivered to consumers. The message behind the recent Newton Developer Evening at MacWorld focused on the momentum that has been building around the platform and the opportunities that have now become so clear: Newton is the leading platform in the PDA marketplace, Apple's licensees are delivering differentiated hardware products to help grow the market, and wireless opportunities are a growing reality. With all of these resources at hand, there has never been a better opportunity for developers to step-up to the Newton platform and become a part of this new market development adventure.

The Newton Developer Evening pulled together the latest news and successes on the platform and offered a kaleidoscopic view of the kinds of opportunities that await developers with the Newton platform. Motorola, one of Apple's Newton Licensees, showcased their new Newton device, the Marco® Wireless Communicator. The Marco device differentiates itself from Apple's MessagePad by focusing on built-in wireless communications solutions. Motorola has opened the door for developers to deliver an unlimited number of solutions to both consumer and corporate marketplaces. Likewise, Apple has delivered a number of communications features for its MessagePad and other Newton devices. Apple's modem set-up package, in conjunction with one of the many cellular-ready PCMCIA cards available on the market, provides developers with the widest array of wireless communications ever. The corporate marketplace will most certainly look to Newton devices to deliver up-to-the-minute data and information from servers and centralized databases to its field sales and off-site work forces. Likewise, consumers will embrace the technology as it readily delivers their latest stock quotes, news and e-mail straight to their fingertips, far away from easy access to phone lines and desktop computers. Other technology companies promise you'll be able to fax from the beach someday. Newton delivers on that promise today.

In addition to the Marco device from Motorola, Apple has also added a new MessagePad to the product line. It offers improved screen clarity and more memory to the user. The MessagePad 120 delivers 2 MB of RAM, leaving the developer more room to increase their applications' capabilities and power. The MessagePad 120 Type II PCMCIA slot now offers up to

continued on page 12

Apple Computer, Inc.
would like to thank Xplain Corporation (the publishers of MacTech™ Magazine)
for lending their expertise in producing the Newton Technology Journal.

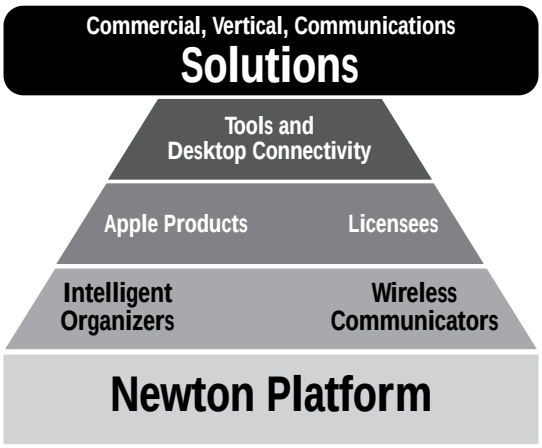
Newton Developer Evening Brings Newton Community Up to Date

by David Glickman, Apple Computer

In the midst of hectic MacWorld schedules and other related events, more than 400 Newton developers gathered for the Newton Platform Developer Evening on January 4th, 1995. Billed as an "update from Apple's Personal Interactive Electronic (PIE) Division and its Newton licensees," the event commenced with an address from Shane Robison, VP and General Manager of the PIE division. Following Robison's talk were presentations on the recently introduced Newton Developer Programs, demonstrations of the new Marco Communicator from Motorola and other key wireless communications technologies, as well as a recap of the PDA market for the past year.

Robison stressed the importance of the Newton developer base, emphatically stating, "We're here today because of you!" He emphasized that the number of applications available for the platform continues to grow and to contribute significantly to the platform's long term success. Today worldwide there are approximately 100 commercial applications, another 300 vertical in-house applications in use or in pilot phases, and 600 shareware titles, ranging from utilities packages to BookMaker-based reference materials.

Robison stressed his commitments to investing in PIE's Developer Relations infrastructure, listening to feedback from the developer community, and continuing to deliver new tools. Elaborating about tools, Robison sketched out developments past the current NTK, specifically mentioning the addition by the Tools team of a new compiler and profiler functionality, with the eventual goal of full C++ access for developers.



In an overview of the Platform Marketing goals of the division, Robison outlined the wide variety of Newton-based products that Apple and its Newton licensees will continue to produce. From intelligent organizers like the MessagePad 120 to the wireless communicator, Marco device, the Newton OS will continue to offer the flexibility necessary for these new devices to evolve. As the diagram below illustrates, the Newton platform is driven by solutions, with tools, cross-platform connectivity, and hardware products combining to meet the demands of the commercial and vertical markets.

Robison concluded his address with a call to action for developers. As momentum for the platform builds and PIE's commitment to delivering more and better tools remains strong, opportunities open for Newton developers.

Motorola's Marco Wireless Communicator

With the introduction of Motorola's Marco® Wireless Communicator, PIE's Newton licensee partners continue to broaden the Newton platform with highly differentiated products. Rick Lane, general manager of the Personal Communicator Product organization at Motorola outlined the Marco Communicator and Motorola's commitment to the Newton platform. The Marco device is based on the latest Newton OS (1.3) and integrates an ARDIS two-way wireless RF modem to deliver an integrated solution for wide area wireless data communications. Users will have access to RadioMail services: store and forward mail, gateway to the Internet, sending faxes, and receiving stock quotes, news, and other information. Marco Wireless Communicator users can also access the ARDIS PersonalMessaging service and send or receive messages with other PersonalMessaging users anywhere within the ARDIS network.

For more information about the Marco Communicator and development opportunities, developers can contact Patrick Pahl, Business Development Manager at Motorola, at the following Internet address: Patrick_Pahl@msmail.wes.mot.com

continued on page 4

Developer Programs

Lee Dorsey, Manager of Developer Programs in PIE, described the expansion of programs for Newton platform developers and laid out the road map for these programs in 1995.

The recently introduced Newton Associates Program offers high quality self-support for Newton developers with access to "developers-only" Newton forums on AppleLink (at AppleLink discount rates), the Newton monthly mailing, and hardware purchase privileges. The Newton Associates Program costs $400 per year.

The Newton Partners Program has been revamped with new offerings at the reduced yearly price of $2500. In addition to all Newton Associates benefits, Partner members will receive expert-level technical support via e-mail, opportunities to participate in select marketing opportunities, and additional hardware purchasing privileges.

The Executive Partners Program is designed specifically for in-house corporate developers. This program offers a designated technical lead from the PIE team, design and code reviews, training, development tools, and executive briefings. This program costs $25,000 per year and is available to a small group of corporate customers.

In outlining Developer Programs plans for 1995, Dorsey announced that support programs will be available worldwide in the Spring of 1995. In addition, Developer Relations is creating new co-marketing opportunities in areas such as solution guides, trade show participation, and press relations.

For more information on Newton Developer Programs, contact Apple's Developer Support Center at DEVSUPPORT@APPLELINK.APPLE.COM or at 408-974-4897. In addition, a full description of the Newton Associates and Partners Programs is published in the February 1995 issue of the Newton Technology Journal.

Wireless Communications Demos

Susan Schuman, Communications Product Manager in PIE, opened the section of communications demonstrations with an overview of the Newton Platform wireless communications strategy. In building upon Newton's built-in software communications capabilities – faxing and NewtonMail – Schuman outlined the current efforts underway to provide tools for the continued development of Newton-savvy communication products.

The Communications group will deliver a modem-enabler package to the entire Newton community (developers and users). In conjunction with modem-setup packages for specific cellular-ready PCMCIA cards, the modem-enabler technology will give users access to a myriad of PCMCIA modem cards for wireless cellular faxing and e-mail.

In a demonstration of PIE's commitment to opening up the Newton platform for new communications developments, several wireless communications demonstrations wowed the crowd. Schuman and Avi Weiss, Communications Engineer in PIE, demonstrated cellular faxing using the Motorola Cellect cellular-ready PCMCIA card. Weiss also sent a page using Ex Machina's Notify software and the Cellect card.

Dayna Communications demonstrated the Serial Roamer RF Packet wireless LAN technology, using a prototype PCMCIA-based Serial Roamer card with a MessagePad. Ed Colby, President of Wayfarer Communications, demonstrated his company's client/server technology for "integrating Newton-based products into enterprise information systems, enabling custom application development" in the areas of data access, forms completion, and mobile communications. With a Windows NT server as the backbone, Wayfarer's technology allows Newton devices to access host information in real time. Colby announced the availability of the Development Kit for Wayfarer's technology for February 1995.

Newton Platform Marketing

Ken Wirt, PIE's Director of Marketing, discussed the marketing strategy for the Apple MessagePad family of products and the current state of the PDA market. The chart below outlines the two distinct markets for the MessagePad: individual users looking for a complete solution for business productivity needs; and vertical markets in which large businesses create custom solutions for key areas such as sales force automation, outside plant management, and healthcare.

	Individual	Vertical
What:	**Handheld device that begins where Organizers end**	**Mobile client for data gathering and access**
Who:	**Mobile Business Professionals**	**Mobile Workforce (Sales Force, Healthcare…)**
Benefit:	• **Instant Access to Critical Info** • **Personal Productivity**	• **Low Cost Solution** • **Mobility**
Advantage:	• **Breadth and depth of solutions** • **Desktop Connectivity** • **Communications solutions** • **Hardware and Software Design**	• **Better tools** • **Price / Performance** • **Communications Solutions**

Wirt also discussed analysts' views of the projected PDA market growth and Newton-based products within that industry. In general, the PDA market is projected to increase approximately 40% in 1995 over 1994 figures, with an installed base estimated at 5 million units in 1999. Currently, Newton-based products have a 60% market share, with a projection to continue to lead the market in the year 2000 with a 40% market share.

In 1995, Wirt sees the most significant growth for the Newton platform in the vertical development area, driven by the emerging wireless technologies, continued developments from Newton licensees, and the increased interest in forms-based technologies for Newton-based devices.

Bringing the Evening to a Close...

The Newton Developer Evening wrapped to a close with Wirt's positive market figures, a brief Q&A, and a few hours of mingling with PIE's staff over bountiful food, drinks and music.

NTJ

© 1994, Apple Computer, Inc. and Calliope Enterprises, Inc.

Stepping through Routing

Introduction

Routing presents a variety of complex implementation tasks. To help clarify these tasks, this article first steps through the basic components of routing and then shows the implementation of four routing elements: beaming, printing, faxing and mailing. By the end of this discussion, the programmer should have a clear understanding of the Newton routing architecture and of what is required to implement routing in an application.

The Routing Frame

Your application's routing frame indicates to the Newton operating system which routing functions it supports, and also ensures that these functions appear in the Action Button picker. The items that you find listed in a typical routing button usually include: Duplicate, Delete, Print, Fax, Mail, and Beam. Each of these items in turn corresponds to a slot in the routing frame:

```
routingFrame := {
  print: …,
  fax: …,
  zap: …,
  mail: …,
  duplicate: …,
  delete: …
}
```

Each of these routing frame slots is itself a frame containing two or three slots of its own. To give you a better idea what this looks like, here is the same routing frame with more of the detail filled in:

```
routingFrame := {
  print: {
    title: "Print Note",
    routeForm: 'printSlip,
    formats: [kFormatSymbol],
  }
  fax: {
    title: "Fax",
    routeForm: 'faxSlip,
    formats: [kFormatSymbol],
  }
  zap: {
    title: "Beam",
    routeForm: 'zapSlip,
   }
  mail: {
    title: "Mail",
    routeForm: 'mailSlip,
    formats: [kFormatSymbol],
  }
  separator: nil,  // dotted line in picker
  duplicate: {
    title: "Duplicate",
    routeScript: 'DuplicateActionScript,
  }
  delete: {
    title: "Delete",
    routeScript: 'DeleteActionScript,
  }
  card: ROM_cardAction,
}
```

The Newton operating system fills in the items in the action picker in precisely the same order that it finds them in this routing frame (making this the singular case in Newton programming where the order, not the names, of the elements in a frame matters). As you can also see in the example above, a slot with a `nil` value specifies a separator line. In this case, the first slot found is `print`. The operating system then looks in the `title` slot or calls the function in the `GetTitle` slot of the print frame to determine what text string to put in the Action button list. Given the above title slots of each action in the routing frame, you would end up with a Action button like this:

Now, starting with the print slot, here are the routing action's slots and their functions:

```
print: {
   title: "Print Note",
   routeForm: 'printSlip,
   formats: [kFormatSymbol],
 }
```

When the user picks the "Print Note" item in the action button, the symbol in the `routeForm` slot is used to determine the slip to open. In this case, the `routeForm` slot contains a reference to the system symbol `printSlip`.

The `formats` Slot

The last slot in the print frame is a `formats` slot. This contains an array of symbols that specify the formatting styles available to the user. In the example, there is only one style, `kFormatSymbol`. These styles are displayed in the format picker that the user sees when selecting an appropriate routing function (for example, mailing, printing, faxing, and so on). Different format pickers may have the same types of formats (see Figure 2) or ones that vary:

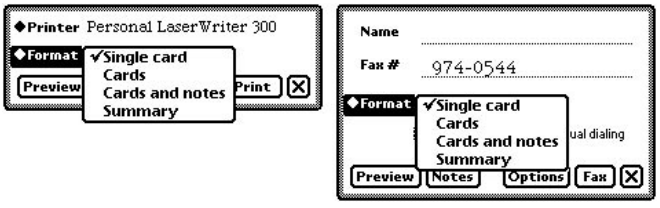


Figure 2

The Other Slots

With the exception of the card slot, the rest of the slots in the routing frame are quite similar in their construction. Here is an overview of the contents of each:

```
fax        routeForm:    'faxSlip          System slip
zap        routeForm:    'zapSlip          System slip
mail       routeForm:    'mailSlip         System slip
delete     routeScript:  'DeleteActionScript
                                           A method you provide
duplicate  routeScript:  'DuplicateActionScript
                                           A method you provide
```

The Application and Its Required Slots

Because the routing frame is typically a slot in your base view template, you will have to ensure that the system knows about your routing capabilities even when the application is not open. Thus, at your application's installation time, its routing frame needs to be installed into the global routing frame. The following line in your `InstallScript` will accomplish this:

```
routing.(kAppSymbol) := packageFrame.theForm.routingFrame;
```

Some of the routing functions also require the addition of certain slots to your application base view. These slots are: `appSymbol`, `target`, `targetView`. These are now described in turn.

appSymbol

This slot contains your application symbol.

target

The `target` slot identifies the frame that is being acted upon by the routing action. It is up to you to maintain the contents of this slot.

targetView

The `targetView` is used by several parts of the Newton system, including routing and filing. When your application registers your routing frame, your base view will actually get routing messages, which, in some cases, include `target` and `targetView` as parameters. Generally, the value of this slot will be your application base view. This view is passed as a parameter to the `routeScript` methods.

Implementing Routing

This section discusses the implementation of four routing actions: beaming, printing, faxing, and mailing. (We assume that your application already supports duplicate, delete, and card actions.) First, you will find a checklist of items needed to support each particular routing action and then you will step through the actual implementation. Note that implementing one routing action makes the next easier; once you support beaming, you will have less to do to implement printing. Let us begin with beaming.

Adding Beaming

To add beaming support to your application, you need to complete the following steps:

The Checklist

1. Add "Beam" to the action picker.
2. Add a title for items that are routed to the In/Out Box.
3. Test sending and receiving an item.
4. Handle putting away an item on a receiving Newton.

1. Add Beam to the Action Picker

Add a new beam slot to the routing frame of your application. This slot should be standard from application to application, so use this code:

```
zap: {
  title: "Beam",
  routeForm: 'zapSlip,
 }
```

If Beam is the first routing action in the frame, then the `title` slot string should be "Beam Object" (where Object stands for whatever is appropriate for your application) instead of just "Beam."

2. Add a Title for Items that are Routed to the In/Out Box

To support titles for items in the In/Out Box you need to add a `SetupRoutingSlip` method to the application base template. Here is an example of this method:

```
func(fields)
begin
  fields.title := info from target && DateNTime(Time());
end
```

This method should create a title slot with a title that is descriptive to the user. For instance, providing the date and information that identifies the target will make it easier for the user to recognize a particular entry in the In/Out box. The title should be no more than 44 characters long; it is used both in the Out Box of the sender and in the In Box of the receiver. If necessary, append an ellipsis character to handle the display of longer strings.

3. Testing Sending and Receiving an Item

After implementing routing in the action picker and adding your `SetupRoutingSlip`, you should be able to beam an item. To test, set the Beam preferences on the receiving Newton to those found in Figure 3. IMPORTANT: Make sure the Inspector is not connected while the beaming occurs.

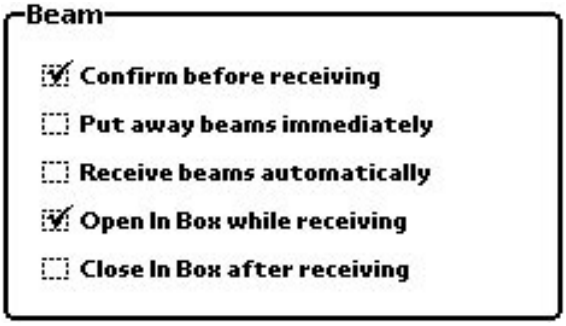


Figure 3

4. Handle Putting Away an Item on the Receiving Newton

To handle putting away an item on a receiving Newton, there are several conditions that you need to take into account:

- A beam can be received when your application is closed.
- A beam can be received before your application has ever been run, and thus your soup may not exist on the receiving Newton.
- A received item may come from a non-existent folder.

Each of these conditions must be handled properly in your `PutAway` method. When a Newton receives a beam and the user selects "Put Away" (or has "Put away beams immediately" set), the `PutAway` message is sent to the application with a matching application symbol. For a trickier approach, a sending application could modify the `appSymbol` slot in the `fields` frame (in the `SetupRoutingSlip` method) and beam to a different application. For instance, our application could beam a name to the Names application.

Now that you have an idea of the type of conditions you need to account for in your method, look at a sample `PutAway`. Remember that `PutAway` is a slot in your application base view:

```
func(item)
begin
  local newEntry := item.body;
  local theSoup := nil;
  local appIsOpen := self.appSoup;

  CheckThatFolderExists(newEntry);
  if appIsOpen then
    theSoup := appIsOpen ;
  else begin
    theSoup := call kRegisterCardSoupFunc with
      (kSoupName, kSoupIndexes, kAppSymbol, kAppObject);
  end;

  theSoup:AddToDefaultStore(newEntry);
  if not appIsOpen then
    call kUnRegisterCardSoupFunc with (kSoupName);
  BroadcastSoupChange(kSoupName);
end
```

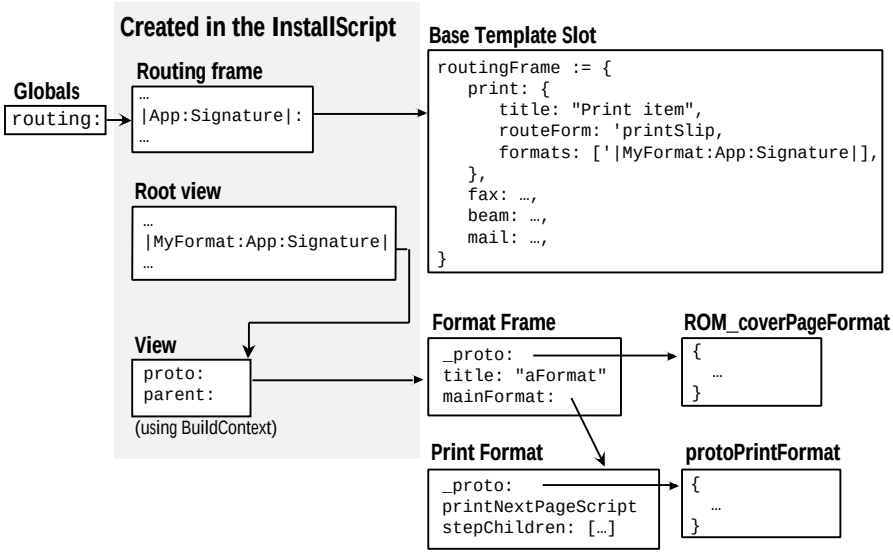


Figure 4

If your application uses soups, then the above method is *fairly* standard. It is worth noting in this method that the `item` frame is roughly the same as the `fields` frame from `SetupRoutingSlip`. Also, `item.body` contains the target from the sending Newton.

The `CheckThatFolderExists` method should only be used by those applications that support folders. Further, it is used to handle a particular type of condition, which is demonstrated by the following case: If Jean sends an item from "Jeans Folder" to Fred's Newton, what folder should the item appear in if Fred doesn't have a "Jeans Folder?" Answer: "Unfiled." The `CheckThatFolderExists` method compares the `labels` slot of its argument to the list of folders on this machine. If it finds it, it does nothing, otherwise it sets the `labels` slot value to `nil`.

Now, look at how the sample `PutAway` code deals with the situation in which the application is not open on the receiving Newton. This application stores the open soup in a slot called `appSoup` which it `nil`s out in the application's `viewQuitScript`. If the application is running, `PutAway` uses that soup. If the application is not running, `PutAway` calls `kRegisterCardSoupFunc` to create the soup (if necessary) on all writable stores (including the default store). Notice that if we called `kRegisterCardSoupFunc`, we must call `kUnregisterCardSoupFunc`.

The `BroadcastSoupChange` function will call the application's `SoupChanged` method, which will redisplay the view if the application is open. If you wanted to be robust in your programming, it wouldn't hurt to have an exception handler in this code as well:

```
try
  theSoup:AddToDefaultStore(…)
onException |evt.ex| do begin
  if not appIsOpen then
    call kUnRegister…
  Rethrow();
end;
if not appIsOpen then
  call kUnRegister…
```

Unfortunately, `AddToDefaultStore` doesn't currently throw an exception, so the exception handler will not execute. In the future, though, `AddToDefaultStore` might throw exceptions and then this code would correctly handle it.

Adding Printing

The Newton implementation of printing is significantly different, though easier, than typical desktop machines. Remember that, within the routing global, your application's routing frame is stored in a slot whose symbol is our application signature. In the routing frame, there is a `formats` slot that is an array. This array contains an entry for each item in the format picker of the print slip. Also, each entry in this array is a symbol. In the root view, each of these symbols is a slot that points to another view. The view uses for its proto a format frame which you normally store in your base template. This is illustrated in Figure 4.

The Checklist

Here is the entire set of steps needed to support printing:

1. Create a print format and add it to your project.
2. Create a format frame in your base template.
3. In your `InstallScript`, call `BuildContext` and create a slot in the root view.
4. In your `RemoveScript`, remove the slot in the root view.
5. Add Print to the routing frame.
6. Modify `SetupRoutingSlip` to save data.
7. Modify `printNextPageScript` in the print format.

Here we describe each of these steps in greater detail.

1. Create a Print Format and Add it to your Project

You create a print format by using the special "New Print Format" layout provided by NTK. The topmost template in this layout needs to use as its proto a `protoPrintFormat`. This template not only has a proto slot but a `printNextPageScript` slot as well. By default, `printNextPageScript` only handles one page, but you can add code to support multiple pages (see step 7).

Once you compile the application, the Print format layout is saved into the package and is accessible by name (for example, `printFormat_nameOfLayoutFile`) at compile time (in an evaluate slot, for instance).

Create any children you wish to have on the printed page. The `protoPrintFormat` view will be sized so that it is the size of the printable area on the page (the page size inset by the margins). You can use justification with child views to place them in appropriate places on the page. For instance, you might want a header at the bottom-right of each page. This could be done with a `protoStaticText` with parent relative bottom and parent relative right justification.

2. Create a Format Frame in your Base Template

The next step is to add a format frame to the application base template. Here is a typical format frame with italics showing the parts you might want to customize:

```
{
  _proto: ROM_coverPageFormat
  title: "Bill",  //text in the format picker
  mainFormat: printFormat_billFormat,
}
```

The `mainFormat` slot points to the `printFormat` you created in the last step. You'll normally create this format frame as a slot in the application base template. Note that you must proto to ROM_coverPageFormat. In addition to providing the coverpage, ROM_coverPageFormat also sets up many other slots that are required for printing.

3. In your InstallScript, call BuildContext and Create a Slot in the Root View

First, create a new constant that the will refer to the application's new print format:

```
constant kFormatSymbol := '|MyPrintFormat:App:Signature|;
```

It is possible for an application to have multiple print formats. They should each have unique symbols. Then the `InstallScript` will write code similar to the following to create a view based on the format frame

using `BuildContext`:

```
InstallScript(partFrame)
begin
  local appBaseTemplate := partFrame.theForm;
  local formatFrame := appBaseTemplate.myFormatFrame;
  GetRoot().(kFormatSymbol) := BuildContext(formatFrame);
  …
end;
```

Remember that `BuildContext` takes a template and creates a view using as its proto that template (its parent is the root view). The view is stored in the root view in a slot with a unique symbol.

4. In your RemoveScript, Remove the Slot from the Root View

Here is where you clean up and remove what is no longer needed.

```
RemoveScript(partFrame)
begin
  …
  RemoveSlot(GetRoot(), kFormatSymbol);
end
```

5. Add Print to the Routing Frame

Now that you have created the necessary layouts and instantiated them, you need to add `Print` to the routing frame. After this step, "Print" will be a choice in the action picker:

```
print: { title: "Print item",
     routeForm: 'printSlip,
     formats: [kFormatSymbol]
   }
```

6. Modify the SetupRoutingSlip to Save Data

For beaming, the target is automatically stored in the `fields` frame. For printing, this is not the case. You need to save whatever information you'll need when you actually print in the `fields` frame. This information may be the target, it may be some information from the target, or something else entirely. Save this information in the `SetupRoutingSlip` method:

```
func(fields)
begin
  fields.title := kAppName &&
    DateNTime(Time()) &&
    distinctive info from target;
  fields.body := information needed for printing/faxing...
end
```

Remember that you only want to save this information for printing; when beaming or mailing, do not use it. You should always assign your data to the `body` slot.

7. Modify printNextPageScript in the Print Format

The `printNextPageScript` message will be sent to your `protoPrintFormat` at the end of each page. If there are no more pages, it should return `nil`. If there are more pages, it should prepare for the next page and return `true`:

```
printNextPageScript : func()
begin
  if moreToPrint then begin
    send messages to children to update their data
    create new children possibly
    remove children possibly
  end else
    return nil;
end
```

When View Messages are Sent During the Printing Process

The way printing works differs depending on the type of printing. For PostScript printers, each view on the page is converted to a PostScript equivalent, and the PostScript representation of the printed page is sent to the printer. For bitmap printers, the Newton images the page into an offscreen bitmap and then sends those bits to the printer. Since there is not enough memory to hold the entire page in an offscreen bitmap, the Newton images the page in horizontal bands from top to bottom. All views that intersect a band are drawn, and the bits for that band are sent to the printer. The offscreen buffer is then used for the next band.

The views involved in printing receive a number of different system messages. At the beginning of printing, the `protoPrintFormat` view (and all of its descendants) receive these messages:

- `viewSetupFormScript`
- `viewSetupChildrenScript`
- `viewSetupDoneScript`

These present good opportunities for the programmer to do any one-time initializations. At the end of each page, the `protoPrintFormat` receives a `printNextPageScript` message. During the processing of each band, each view that intersects the band is also sent the `viewDrawScript` message.

From any of these methods, you can access the `fields` frame (via parent inheritance). Thus, you can access any information you saved in the `fields` frame in the `SetupRoutingSlip` method.

Supporting More Than One Print Format

If you want to support more that one type of print format, simply create multiple print formats and add them to your project. Then create multiple format frames in your application base template. Build these multiple format views in your `InstallScript` instead of just the original one. In the `RemoveScript`, remove the slots you created in the `InstallScript`. Your last duty is to create special format symbols for each of the format frames and then add the multiple symbols to the `formats` slot in the routing frame.

Faxing

Here is the good news – once you've implemented printing, you have already gone a long way towards supporting faxing. To support faxing, you need a separate entry in the routing frame, and you need to make sure your imaging is quick enough to avoid timing out. Here is a checklist for faxing:

Faxing Checklist

1. Add faxing to the routing frame.
2. Verify that your page images quickly.

1. Add Faxing to the Routing Frame.

```
fax: {
     title: "Fax item",
     routeForm: 'faxSlip,
     formats: [kFormatSymbol]
  }
```

Any format frame that supports printing will also support faxing. For both faxing and printing, a format frame must use as its proto `ROM_coverPageFormat`. The name of this proto is somewhat misleading since this proto is also necessary for printing.

2. Verify that Your Page Images Quickly.

Faxing requires careful attention because of the very real possibility of timing out in the middle of a fax. It is imperative that the actual sending of a fax be speedy. Fax machines will timeout if approximately 5 seconds elapse without receiving information. Thus, once the connection has been made,,, everything must proceed at a fairly brisk pace. Further, each band must image quickly because all imaging occurs while the fax connection exists. As a result, you will need to do the following:

- Use the `viewSetupFormScript` of the `printFormat` for expensive operations–before calling `inherited:viewSetupFormScript()`.
- Make `printNextPageScript` & `viewDrawScript` quick.
- Consider using multiple views with height less than the entire page (only views that intersect a band need to be drawn).

These considerations are especially necessary if you have complicated pages to fax. For many applications, however, all that is required is some testing that demonstrates that faxing doesn't time out. Note that you should test this on either a StyleWriter printer, a real Fax machine or some other type of rasterizing device. When printing to a PostScript printer, the page is imaged as one band, so this does not provide a good test.

Mailing

Mailing involves some additions to the format frame to support the creation of text that will be sent.

The Checklist

1. Add mail to the routing frame.
2. Add a method to the application that will create text to send.
3. Add a text slot to the format frame that references this new method.
4. Support closures by adding an `attachment` slot to the format frame.

NTJ

Newton®

To send comments or to make requests for articles in Newton Technology Journal,
send mail via internet to: piesysop@applelink.apple.com

Newton®

To request information or an application on Apple's Newton developer programs,
contact Apple's Developer Support Center
at 408-974-4897
or Applelink: DEVSUPPORT
or Internet: devsupport@applelink.apple.com.

Apple Announces New Newton MessagePad 120, Communications and Software Solutions!

by Patty Tulloch, Apple Computer, Inc.

Apple Computer, Inc. has announced the newest member of their Newton MessagePad family of products, the MessagePad 120. This easy-to-use, hand-held device provides convenient access to the information you need, no matter where you happen to be — in a meeting, at home, or on the road.

Improved Design and Screen Clarity

Like the MessagePad 110, the new MessagePad 120 has a slimline design, so it fits easily into the palm of your hand. The protective lid is now removable so that you can customize the MessagePad to fit your usage pattern and the PCMCIA card lock has been moved to the side, to make it more accessible when you have your lid attached. In addition, the new MessagePad 120 screen offers reduced shadowing and better screen clarity.

More Internal Memory

One of the most significant MessagePad 120 feature enhancements is the new 2MB internal memory configuration. This offers users three times more internal space than is available in the current MessagePad 110. Users will be able to put more of their applications in internal memory, thus freeing up their PCMCIA slots for add-on communications cards and peripherals. In addition, the total configuration cost will be significantly reduced, as most users will not have to invest in extra memory cards. The new memory configuration offers a combination of RAM and Flash storage, assuring that even if a user accidentally removes all the batteries, their data will be saved.

Support for High-Powered PCMCIA Cards

The MessagePad 120 Type II PCMCIA slot now offers up to 325 mA capacity, making it better suited for development of high-powered communications cards and peripherals. The PCMCIA slot accommodates a range of add-on Newton software application cards including: LAN-line, cellular data and fax modem cards; paging cards; memory cards; and others.

1.3 (344311) System Software Improvements

Improvements have been made to the current 1.3 version of system software, making it more reliable and intuitive. Some of the more significant enhancements are outlined here.

- Notification windows (alarms) are no longer closed when the Newton goes to sleep. This will prevent users from missing notifications that occur when the Newton is unattended.
- The '*' and '#' characters can be used when dialing phone numbers from the call slip. For example, a user could set the dialing prefix to '*70' to turn off call-waiting, if their phone system allows it.
- A recognition problem which could have caused the accuracy of handwriting recognition to gradually degrade for some users has been corrected.
- Switching to guest mode and back now restores the owner's letter styles settings, when the writing style is set to "Printed Only".
- A modem-dialing problem for users with location set to a city in Japan has been corrected. The Newton now sends the correct modem initialization string "AT%J&P1".
- A problem that could have put the Newton into a state in which PCMCIA cards could be recognized without a soft reset has been corrected.
- The reliability of erasing flash PCMCIA memory cards has been improved.

New Communications Solutions

With the MessagePad 120, you can communicate in several ways. Using an external modem or the PCMCIA modem, you can fax documents anywhere, send and receive e-mail messages with people who subscribe to services such as NewtonMail, CompuServe, and other Internet services.

With a cellular phone, software, cable and a new PCMCIA FAX/modem, you can now also fax documents, and send and receive e-mail messages wirelessly.

You can turn your Message Pad 120 into a paging device with the optional Apple Mobile Messaging system bundle. This includes a PCMCIA Paging Card from Socket, software from ExMachina and a subscription to Apple's notification service.

New Software Solutions

Several business, information management, and reference software packages have been announced in conjunction with the introduction of the MessagePad 120. These include:

- *ACT!*, from Symantec, distributed by StarCore
  This is contact manager software that puts vital, up-to-date client information at your fingertips, organizes your schedule, automates your record-keeping and correspondence, and synchronizes with your Windows and Macintosh desktop ACT! applications.
- *Berlitz Five Language Interpreter* from StarCore
  This application translates 20,000 words and 15,000 phrases between English, French, Spanish, Italian and German
- *Newton Utilities* from StarCore

This package includes four applications with unique features to maximize memory, manage software and help you work faster. The Shortcuts application allows you to create a customized floating palette of your favorite features.

- *PowerForms*™ from Sestra, Inc.
  This application turns your Newton into a mobile electronic briefcase containing business and personal forms for professionals on the go. Examples include purchase orders, invoices, sales quotations, shop orders, sales records, employee personnel files, advertising analysis, and so on.
- *The Newton Enhancement Pack* from StarCore
  This bundle features three great applications on one 2 MB PCMCIA flash card. The applications are Graffiti, Newton Utilities, and Action Names.
- *PocketQuicken* from Intuit
  This facilitates tracking all your finances while you're on the go. With Pocket Quicken, you can instantly update your checking, savings, credit card, and cash accounts. You can also track business expenses by trip, project, and client.
- *CIS Retriever* from BlackLabs
  The Newton client for CompuServe, this allows users to send and retrieve mail, and to read and post Forum messages. This software supports CIS Xtenders which provide access to stock quotes, airline information, business databases, and so on.

NTJ

To request information or an application on Apple's Newton developer programs,
contact Apple's Developer Support Center
at 408-974-4897
or Applelink: DEVSUPPORT
or Internet: devsupport@applelink.apple.com.

continued from page 2

Letter From the Editor

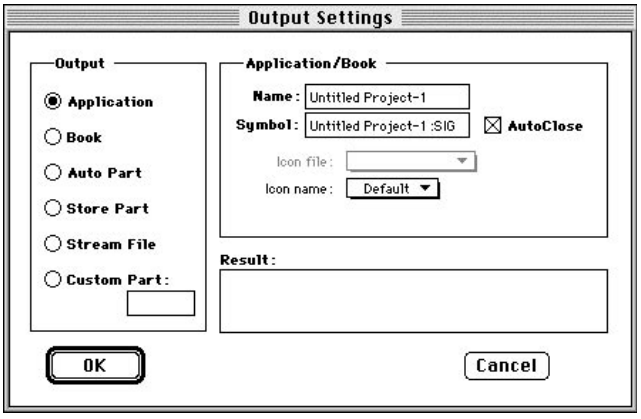
325 mA capacity, making it better suited for high-powered communications cards and peripherals.

The message that is coming across about Newton is impossible to ignore. With three major companies currently manufacturing and selling Newton devices, the Newton platform continues to evolve. More Newton licensees are gearing up to introduce even further differentiated products to a growing customer base in 1995. Any application written for the Newton platform will run on any of these Newton devices, allowing developers to continually broaden the potential customer base for applications.

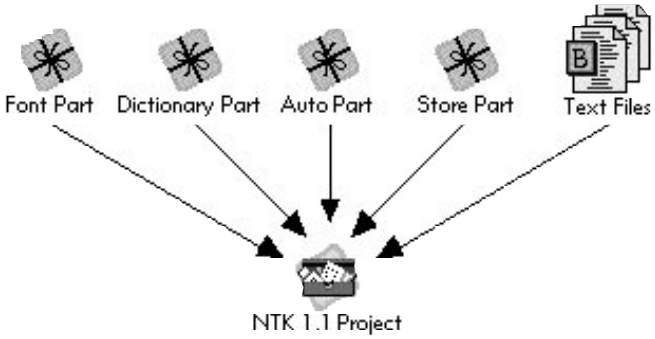
As Newton devices proliferate in the corporate marketplace and find their way into the hands of professionals on-the-go, the Newton platform offers developers the opportunity for greater sales of existing products, new business markets, and unlimited growth potential. As Philip Ivanier, Manager of the Newton Developer Relations Group, said, "we're constantly improving the palette of paints available for developers." With such a palette at your disposal, our hope is that one or more of you will be the developer who builds the next "killer application" or broad-based solution that becomes an essential part of every user's Newton!

continued from page 1

Coming Soon: Newton Toolkit 1.5



Create a variety of part types with the new Output Settings controls.



In addition to the standard files, projects now support multiple text files and parts.

NTK 1.5 projects support layouts, resources, text files, book files, and parts. This means your project can contain multiple text files. You can also build multi-part packages and specify build order to suit your needs.

A new button bar in the Inspector offers controls of the new Profiler tool and standard debugging functions.

Introducing NewtonScript Performance Tools

To help developers study the performance of their applications, NTK 1.5 features a new performance profiling tool. The Profiler generates an analysis of your functions and system calls. Using the new Inspector button bar, you can upload results in the form of a table. The report includes useful information such as time elapsed per function, number of times a function was called, and size of each function in memory. This can often alert you to bottlenecks in your code that may need to be reworked or optimized.

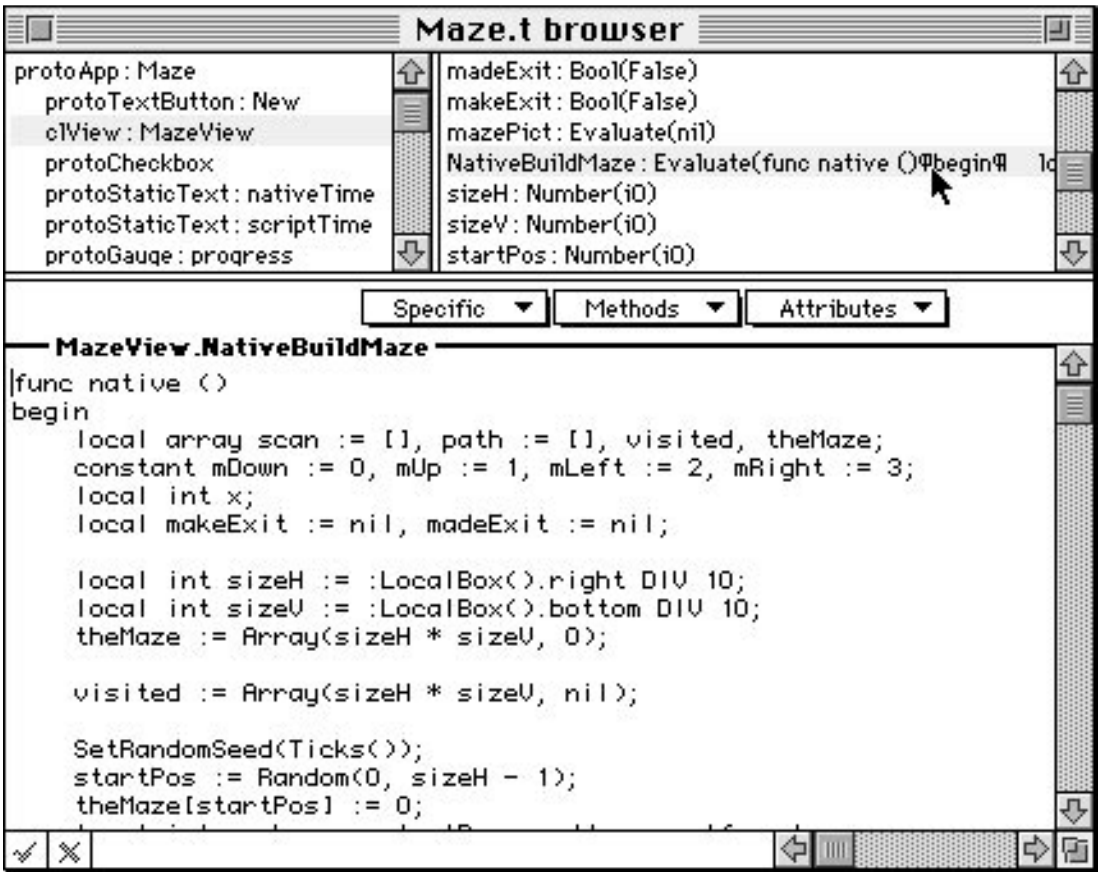
Optimize With the Native NewtonScript Compiler

For stand-alone code that does not interact with Newton subsystems (that is, soups and views) you may consider optimizing to native ARM code. While Native NewtonScript is much larger than interpreted NewtonScript, it retains portability and can deliver higher performance. With NTK 1.5, you can make this trade-off on a function-by-function basis.

Function	Min	Max	Average	%	Entries	Size
MainLayout.TopView.CalculateDistance	500	900	600	7	98	250
MainLayout.TopView.SortNames	300	10000	900	10	515	780
ReportLayout.Report View.CompareResults	600	988	948	10	43	2500
ProjectData.FindPath.func1	477	500	490	3	2	5680

Quickly upload statistics to analyze the performance of your functions.

continued on page 14



Use the "native" keyword to optimize a particular function.

However, using the Native compiler is not a simple operation; you'll want to read the documentation carefully. There are many hints, such as explicit typing, that you must consider to ensure that compilation results in a performance boost. Native NewtonScript may not solve every application's performance problems, but it's an important step. We are committed to delivering both high-level and low-level tools to enable developers to fully recognize Newton's potential.

BEYOND NTK 1.5

Our goal is to provide great tools with which to build great applications. Newton Toolkit 1.5 will deliver greater productivity as well as powerful new capabilities. We look forward to bringing you even better Newton tools in the future. Please send comments and feature requests to NTKBUGS@NEWTON.APPLE.COM.

NTJ

If you have an idea for an article you'd like to write for Newton Technology Journal, send it via internet to: piesysop@applelink.apple.com or AppleLink: PIESYSOP

continued from page 1

NewtonScript Performance Tuning

Some Resources

There are two books that contain particularly good discussions of optimizing code. The first is devoted solely to optimization, while the second covers all aspects of programming:

Jon Bentley, *Writing Efficient Programs*, Prentice Hall, 1982, ISBN 0-13-970244-X
Steve McConnell, *Code Complete*, Microsoft Press, 1993 ISBN 1-55615-484-4

Both books correctly point out that optimization should be done after the code is written. You measure the performance of an application, determine where the application is spending most of its time, and then concentrate on optimizing those areas.

Measuring Performance of a Newton Application

For the Newton, the only real way to measure performance is to call `Ticks()` before and after a piece of code. This lets you see how long the code took to execute. The `Ticks()` function, however, returns a number in 60ths of a second, To measure quick operations, you may want to do them repetitively in a loop.

The following is generally useful:

```
theRoutine := func(args)
begin
  …
end;
frame := {result: nil, time: nil};
oldTicks := Ticks();
frame.result := call theRoutine with (theArgs);
frame.time := Ticks() - oldTicks;
frame
```

To do this from the Inspector, you will need to embed the timing code in a function. This way the code can include loops (the Inspector doesn't allow loops at the top level).

Here's an example of executing this:

```
theRoutine := func()
begin
  local total := 0;
  for i := 1 to 1000 do begin
  end;
  return total;
end;
frame := {result: nil, time: nil};
oldTicks := Ticks();
frame.results := call theRoutine with ();
frame.time := Ticks() - oldTicks;
frame
#441AEF1  {result: 0,
       Time: 8}

theRoutine := func()
begin
  local total := 0;
  for i := 1 to 1000 do begin
    total := total + i;
  end;
  return total;
end;
frame := {result: nil, time: nil};
oldTicks := Ticks();
frame.results := call theRoutine with ();
frame.time := Ticks() - oldTicks;
frame
#441AEF1  {result: 500500,
       Time: 14}
```

Notice that there are two timings here. The first measured 1000 iterations of an empty loop, and the second measured 1000 iterations of a loop with an integer addition each iteration. The difference between the two is 6 ticks, or 1/10 of a second. Thus, a rough estimate of the speed of an addition of two local variables is 1/10000 of a second.

Speeding Up Your NewtonScript

Since NewtonScript is interpreted byte-code (at least until NTK provides a NewtonScript compiler), the best way to increase NewtonScript speed is to use raw NewtonScript as little as possible.

Use Built-in Functions in Preference to Writing Your Own.

For example, if you want to sort an array, rather than writing your own sorting routine similar to:

```
call kMySortFunc(anArray)
```

you are better off calling the built-in `Sort` function. Since the built-in global functions are written in C++, they run much faster than your custom NewtonScript code. Using a built-in function also has the advantage of reducing the size of your package.

You can see a continuation of this idea in how you handle certain NewtonScript functions. Some functions themselves take functions as parameters. For instance, `Sort` takes a function as the second parameter. Assume you have an array of frames, each of which has a `height` slot, and you wish to sort the frames in increasing order of height. One way to do this is:

```
Sort(myArray, func(a, b) return a.height -  b.height, nil);
```

However, the third parameter of `Sort` is a symbol specifying the slot on which to sort. Therefore, the example can be rewritten as:

```
Sort(myArray, func(a, b) return a - b, 'height);
```

This should be slightly quicker, since the two accesses of the height slot are now made in C++ rather than in NewtonScript. A sample with 500 elements yielded 884 ticks for the first case, 810 for second).

Even better, however, is to avoid NewtonScript entirely (even `return a - b`). This can be done with `Sort` by passing the symbol `'|<|` as the second parameter:

```
Sort(myArray, '|<|, 'height)
```

For the same sample, this is substantially faster, taking only 42 ticks, a 20 times speedup.

Always Declare Your Local Variables

There are several reasons for this rule, not just performance implications. One reason is code readability. If you define your locals using the keyword, then readers of your code know what are local variables, and what are globals or inherited slots. Additionally, at some point, NewtonScript may require declaration of local variables, so you might as well be prepared. The last reason for following this rule is that there is no good reason not to.

Speed is the most important factor, however, for declaring locals. When the NewtonScript compiler sees the use of a declared local variable, it generates code that accesses the variable via a numeric stack offset. On the other hand, the code to access variables not declared locally requires a dynamic lookup based on the variable name. This dynamic lookup is slower.

The situation when assigning to a variable for the first time is even worse. If a variable isn't declared local, the assignment requires a lookup, starting first with a lookup as a local, then using both proto and parent inheritance, and finally as a global. If the variable isn't found at this point, the interpreter creates the variable as a local.

The following code, called with an array of 1000 elements, takes 66 ticks:

```
SumArray := func(anArray)
begin
  total := 0;
  foreach elem in anArray do
    total := total + elem;
  return total;
end;
```

By simply declaring the local variables, the time is reduced to 31 ticks:

```
SumArray := func(anArray)
begin
  local total := 0;
  foreach elem in anArray do
    total := total + elem;
  return total;
end;
```

Loop Variables and Local Declarations

Loop variables are automatically declared as locals. In the following code, the variables in bold need not be explicitly declared as locals:

```
for i := 1 to 100 do
  …
foreach elem in array do
  …
foreach slotSymbol, value in frame do
  …
```

Using foreach Correctly

It is quicker to iterate through the elements of an array with `foreach` than it is with an explicit `for` loop. Here are two examples:

```
SumArray1 := func(anArray)
begin
  local total := 0;
  foreach elem in anArray do
    total := total + elem;
  return total;
end;

SumArray2 := func(anArray)
begin
  local total := 0;
  for i := 0 to Length(anArray) - 1 do
    total := total + anArray[i];
  return total;
end;
```

With `foreach`, you have a specialized NewtonScript construct which the interpreter can handle quickly. It is also easier to use, since you can't accidentally get the starting or ending conditions wrong. In this case, `SumArray1` took 18 ticks, while `SumArray2` took 32 ticks (for a 1000-element array).

Even with NewtonScript constructs like `foreach`, measurement is important. There are times when using it is slower. For example:

```
ZeroArray1 := func(anArray)
begin
  foreach index, elem in anArray do
    anArray[index] := 0;
end;


ZeroArray2 := func(anArray)
begin
  for index := 0 to Length(anArray) - 1 do
    anArray[index] := 0;
end;
```

Although `ZeroArray1` is easier to read, the interpreter sets the value of `elem` (which is unused) as well as the value of `index` each time through the loop. Here, `ZeroArray1` took 42 ticks when called with a 1000-element array, while `ZeroArray2` took only 15 ticks.

Function calls are expensive

Compared to some other languages, the cost of calling a function in NewtonScript is expensive.

The Newton can make not quite 1000 function calls per second (if the functions do nothing, and no lookup is necessary to call them). The following code took between 60 and 70 ticks to execute:

```
g := func()
begin
  local f := func()
    begin
    end;

  Print(Ticks());
  for i := 1 to 1000 do
    call f with ();
  Print(Ticks());
end
```

Sending a message is somewhat more expensive, mostly due to the cost of looking up the method in the frame to which the message is sent.

Cache expensive calculations

Currently, the NewtonScript compiler does no common sub-expression elimination optimization. Every expression in your code will be evaluated. This means that if you refer to an inherited variable many times in a method, the lookup for that slot must occur many times. Consider caching the results of operations.

For instance, instead of:

```
Method := func(s)
begin
  for i := 1 to 100 do
    s := s + inheritedVar;
  return s;
end;
```

use:

```
Method := func(s)
begin
  local cachedVar := inheritedVar;
  for i := 1 to 100 do
    s := s + cachedVar;
  return s;
end;
```

The latter code avoids 999 variable lookups.

Along similar lines, if you are accessing nested frames repeatedly, consider caching the result of the dot lookup. Rather than:

```
for i := 1 to 100 do
  a.b.c.d[i] := i;
```

use:

```
local arr := a.b.c.d;
for i := 1 to 100 do
  arr[i] := i;
```

The latter code took 6 ticks while the former code took 7 ticks.

Eliminate loop Invariants

A common compiler optimization is to move expressions whose values don't change outside of a loop. At this point, however, NewtonScript doesn't do this optimization, so you must do it yourself. For example, rather than:

```
for i := 1 to 100 do
  :Method(x + y + 53 * z, 2 * i);
```

use:

```
local value := x + y + 53 * z;
for i := 1 to 100 do
  :Method(value , 2 * i);
```

By making this change, the calculation of `x + y + 53 * z` is done only once, rather than 100 times.

Using and Allocating Memory Efficiently

Efficiently handling memory within a NewtonScript application involves both attention to memory allocation and to whether you are using RAM or ROM. First, you must deal with the more simple issue of measuring your memory use.

Measuring Memory Use

In order to measure space in your program, the best bet is to call `Stats()`. This method returns the total amount of memory available. Since `Stats` doesn't take into account memory that needs to be garbage collected, however, you should always call `GC()` first.

Here is some code to use to see how much memory a function takes:

```
theRoutine := func(args)
begin
  …
end;
frame := {result: nil, space: nil};
GC();
oldSpace := Stats();
frame.result := call theRoutine with (theArgs);
GC();
frame.space := oldSpace - Stats();
frame
```

Here is an example of its use:

```
theRoutine := func()
begin
  return [1, 2, 3, 4, 5, 6, 7, 8]
end;
frame := {result: nil, space: nil};
GC();
oldSpace := Stats();
frame.result := call theRoutine with ();
GC();
frame.space := oldSpace - Stats();
frame
Free: 38504, Largest: 50616
Free: 38472, Largest: 50568
#4414C39  {result: [#4414C51],
       space: 48}
```

The above example shows that this eight-element array takes 48 bytes. (See the NewtonScript Q&A for a detailed discussion of object sizes).

Efficiently Allocating Memory

Allocating memory takes time. In addition, it uses RAM – a very precious resource on the Newton. For these reasons, it is best to avoid allocating memory whenever possible. Here are some ways to do that:

Using Constant Frames and Arrays

If you use an array or frame that will not change at run time, make sure that it is constant. For example, there is a substantial difference between:

```
myFrame := {a: 3, b: 4}
```

and:

```
kMyFrame := '{a: 3, b: 4}
```

The former is technically a frame constructor that allocates memory from the frame heap for the frame at run time. The second frame is created at compile time and is stored as part of your package. This trade of RAM for ROM storage will always be a better programming method.

There is also a syntax you should use for defining constant arrays:

```
'[1, 2, 3]
```

You can also use `DefConst` to create constants:

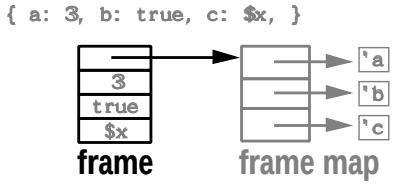
```
DefConst('kMyFrame, {a: 3, b: 4 * 5});
```

In the current implementation of NewtonScript, strings literals are constant. Therefore, the following code creates a string which is stored in the package, rather than in the frame heap:

```
s := "abcd";
```

Sharing Frame Maps

As shown in the following diagram, frames created with the same frame constructor share the same frame map. The frame map contains the mapping from slot symbol to location within the frame.



To ensure that you use the same frame maps, create each frame with the same frame constructor. For example, if you need to create a bounds frame, don't use:

```
{left: 10, top: 5, right: 30, bottom: 100}
```

Instead, call `SetBounds`

```
SetBounds(10, 5, 30, 100)
```

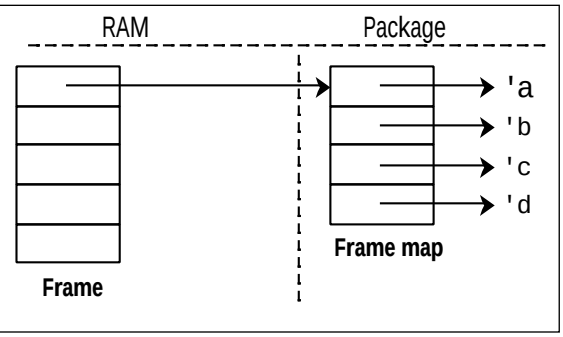
Further, with `SetBounds`, the frame map of the rectangle will be in the Newton's ROM, rather than in your package. You can also use other system functions that return rectangles such as RelBounds.

On a similar note, you should ensure that your frame constructor contains all the slots your frame needs, rather than adding new slots dynamically. Adding a new slot not only requires allocating memory for the slot value (an unavoidable four bytes), but also requires allocating memory for the frame map entry. By making sure the initial frame constructor contains the necessary slots, the frame map can be in the package, rather than in the frame heap.

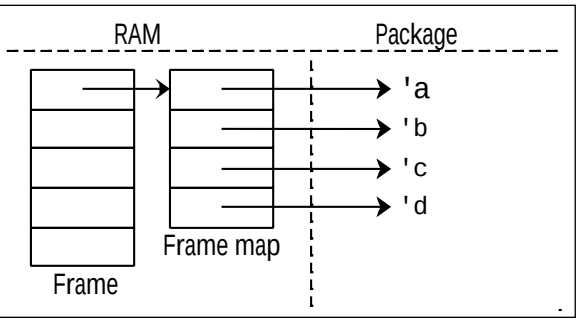
Here are two routines which each return a frame.

The first creates all the slots in the frame constructor: The second creates each slot dynamically:

```
MakeFrame := func()
begin
     local frame :=
           {a: nil, b: nil,
            c: nil, d: nil};
     frame.a := 1;
     frame.b := 2;
     frame.c := 3;
     frame.d := 4;
     return frame;
end;
```




```
MakeFrame2 := func()
begin
     local frame := {};
     frame.a := 1;
     frame.b := 2;
     frame.c := 3;
     frame.d := 4;
     return frame;
end;
```



The first routine uses 32 bytes of the frame heap, while the second uses 64 bytes. This first routine is not just smaller, it is also faster. It takes 17 ticks to call the first routine 100 times, while it takes 32 ticks to call the second one 100 times.

Don't Blindly use EnsureInternal, TotalClone, or DeepClone

Remember that NewtonScript is specifically designed to share memory. When you use `EnsureInternal`, `TotalClone`, or `DeepClone`, you are making copies so that memory is not shared. Don't do this unless you have a good reason. Generally speaking, it is important to be aware of what these three routines do, and only use them when necessary.

Summary

It should now be obvious that, by optimizing a few key areas in your code, you can both increase the responsiveness of your program and use memory much more efficiently. Having good optimizing techniques in the first place can only help. Beyond that, rely on built-in functions rather than creating your own custom NewtonScript. Remember to define your locals, use the right loop for the job, and define constants where ever possible.

You can save memory in your programs by paying attention to details such as frame maps. You should also never blindly use the cloning functions to do something that can be handled in another way. Finally, your awareness of how operators use memory can help you determine when to change to a more efficient method.

NTJ

Apple Developer Group

Newton Developer Programs

Apple offers two programs for Newton developers—the Newton Associates Program and the Newton Partners Program. The Newton Associates Program is a low cost, self-help development support program. The Newton Partners Program is designed for developers who need expert-level development support via electronic mail. Both programs provide focused Newton development information and discounts on development hardware, software, and tools—all of which can reduce your organization's development time and costs.

Newton Associates Program

This program is specially designed to provide low-cost, self-help development resources to Newton developers. Participants gain access to online technical information and receive monthly mailings of essential Newton development information. With the discounts that participants receive on everything from development hardware to training, many find that their annual fee is recouped in the first few months of membership.

Self-Help Technical Support

- Online technical information and developer forums
- Access to Apple's technical Q&A reference library
- Use of Apple's Third-Party Compatibility Test Lab

Newton Developer Mailing

- *Newton Technology Journal*
- *Newton Developer CD*, which may include:
  - Newton Sample Code
  - Newton System Software
  - Newton tools and utilities
  - Marketing and business information
- *Apple Directions—The Developer Business Report*

Savings on Hardware, Tools, and Training

- Discounts on certain development-related Apple hardware
- Apple Newton development tool updates
- Discounted rates on Apple's online service
- US $100 Newton development training discount

Other

- Developer Support Center Services
- Developer conference invitations
- *Apple Developer University Catalog*
- *APDA Tools Catalog*

Newton Partners Program

This expert-level development support program helps developers create products and services compatible with Newton products. Newton Partners receive all Newton Associates Program features, as well as programming-level development support via electronic mail, discounts on five additional Newton development units, and participation in select marketing opportunities.

With this program's focused approach to the delivery of Newton-specific information, the Newton Partners Program, more than ever, can help keep your projects on the fast track and reduce development costs.

Expert Newton Programming-level Support

- One-to-one technical support via e-mail

Apple Newton Hardware

- Discounts on five additional Newton development units

Pre-release Hardware and Software

- Consideration as a test site for pre-release Newton products

Marketing Activities

- Participation in select Apple-sponsored marketing and PR activities

All Newton Associates Program Features:

- Developer Support Center Services
- Self-help technical support
- Newton Developer mailing
- Savings on hardware, tools, and training

For Information on All Apple Developer Programs

Call the Developer Support Center for information or an application. Developers outside the United States and Canada should contact their local Apple office for information about local programs.

Developer Support Center at (408) 974-4897
Apple Computer, Inc.
1 Infinite Loop, M/S 303-1P

Newton

© 1994, Apple Computer, Inc. and Calliope Enterprises, Inc

Advanced Debugging

by Julie McKeehan and Neil Rhodes, Calliope Enterprises, Inc.

This article discusses several little-known debugging techniques which should be part of the arsenal of all Newton programmers. These techniques include:

- Replacing procedures on the fly
- Overriding global functions
- Overriding root view methods
- Making trace work the way you want it to on a particular piece of code

Replacing Procedures on the Fly

You are probably familiar with the common debug cycle on the Newton:

- You compile and download
- You run the application looking for an error
- You then debug the error using the Inspector or ViewFrame (or a combination of both)

When a bug is found in a method, the straightforward approach is to fix it, and then start the whole process over again: compile, download, run the application, and test to make sure the fix is correct. The good news about this approach is that this compile/download/run cycle is much quicker than on many platforms (e.g., Macintosh Programmer's Workshop). The bad news, as any programmer will tell you, is that this cycle just never is quick enough.

A speedier approach is available. You can use the Inspector to replace the method that has the error while still running the application. For example, let's say you have a template named "myButton" which has a method named `Method1`:

```
func(a, b, c)
begin
  local x := 1;
  local y := 1;
  local z := 1;

  if a then begin
    y := y * 2;
    z := :Method2(a, b, c, x, y);
  end;
  return x * y * z;
end
```

Imagine that you have a problem with `Method1`, and you would like to print out its parameters. You can do this by executing the following in the Inspector:

```
Debug("myButton").Method1 := func(a, b, c)
begin
  Print(a);
  Print(b);
  Print(c);
  inherited:Method1(a, b, c);
end
```

When you execute this method, it is compiled from the Inspector and stored in the frames heap of the Newton. The `Method1` slot is now part of the `myButton` view, and now whenever the `Method1` message is sent to `myButton`, the new overridden version will execute instead. Since part of the new method is to print out the parameters, it will do so and then call the old version.

Notice that you could even completely replace `Method1`:

```
Debug("myButton").Method1 := func(a, b, c)
begin
  local x := 3;
  local y := 3;
  local z := 5;

  if a then begin
    BreakLoop();
    y := y * 2;
    z := :Method2(a, b, c, x, y);
  end;
  return x * y * z;
end;
```

The one requirement for this technique to work is that the view to which you are adding a method must exist. Thus, you cannot add code to a closed view (unless, of course, it is declared to a view that is still open).

You can use this approach to add code incrementally to a view — one method at a time — thus making debugging much simpler. The only tricky part is remembering to copy the finished product back into your project once you're satisfied that the code is correct. The best code in the world won't help your application if it is left sitting in the Inspector window.

Overriding Global Functions

There are times when you will find it useful to replace a global function. For example, you may want to know when a particular function is called, or what its parameters are. As an example of this, let's say you want to find out when (or even whether) a particular application calls `BroadcastSoupChange` and what soup name it passes when it does.

Here is how you perform this neat little trick. First, you create a global function that references the old global function (usually, you'll want to augment the behavior of a function, not completely replace it). Remember that all global functions are stored in the global variable `functions` frame, which is itself a global variable. From the Inspector, execute this code:

```
functions.OldBroadcastSoupChange :=
functions.BroadcastSoupChange;
```

Second, create a new function that takes the same number of parameters and that calls the old function:

```
functions.BroadcastSoupChange := func(s)
begin
  Print("BroadcastSoupChange called with" && s);
  OldBroadcastSoupChange(s);
end;
```

Overriding Root View Methods

Another technique that can be very useful is being able to override root view methods for debugging purposes. Methods like `Notify`, `FindSoupExcerpt`, `Open`, and `Close` are all root view methods that at times might be useful to override for debugging purposes.

The technique is very similar to the one for replacing methods in your own application. Simply add a slot to the root view that overrides the method found in the root template. All you have to do is execute the following code from the Inspector:

```
GetRoot().MethodToOverride := func(params)
begin
  whatever you want to do
  inherited:MethodToOverride(params);
end;
```

Making Trace Apply Immediately

The `trace` global variable controls tracing and it would probably be very useful if you could precisely control when it turns on and off. And as many programmers have learned, simply setting it to `true` (or `'functions`) for your whole program, only causes enormous amounts of information to be printed to the Inspector. Given this bulky bitstream, many programmers would like to be able to set the variable to `true` only around a particular section of code and then be able to examine only that output.

While you would like it to be, it's not as easy as surrounding your code with the following:

```
trace := true;
code to trace
trace := nil;
```

The reason this doesn't work is that the NewtonScript interpreter only occasionally looks at the value of the `trace` variable. Once you think through the logic of it, you will realize that this is actually a good thing; if the interpreter checked the value on every instruction, the speed of the interpreter would be severely slowed for all Newtons. Since slowing the Newton down is not a choice that anyone wants, the NewtonScript interpreter irregularly checks the value of this variable when a function is called (function calls are expensive enough that one check of a global variable has no substantial impact). Of course, this does leave you with the problem of how to trace around just a little bit of code.

Happily there is a nice solution to this problem. To cause tracing to happen around a particular section of code, all you have to do is make sure to call a function immediately after setting the `trace` variable:

```
trace := true;
Apply(func() nil, []);
code to trace
trace := nil;
Apply(func() nil, []);
```

Simply by executing an empty function using `Apply`, we've caused the interpreter to check the `trace` variable and start tracing before the code of interest is executed. Similarly, as soon as the code of interest completes, we turn tracing off.

There is one small proviso that accompanies this trick and that is that you can not depend upon it to work in the future versions. The particular manner in which the `trace` global variable is checked may well be different in the future.

Summary

One or more of these techniques may be of use to you in your Newton debugging. Remember that all the techniques that create functions in the frames heap are only for debugging and will be lost when the Newton is reset. If you want to keep the results of your carefully crafted code, then copy it back into your project. These techniques are very useful, however, when you want to quickly see the effect of different code without expensive recompile and download cycles. NTJ

To send comments or to make requests for articles in Newton Technology Journal, send mail via internet to: piesysop@applelink.apple.com

© Copyright 1994 Motorola, Inc.

Marco Software Architecture, Version 1.0

by Dave Baum, Motorola, Inc. Wireless Data Group

Basic System Architecture

The Marco communications architecture is shown below in figure 1. Some of the features provided for in the architecture include sharing the radio among multiple clients, multiple mail transport services, wireless and wireline mail applications, and several different developer APIs.

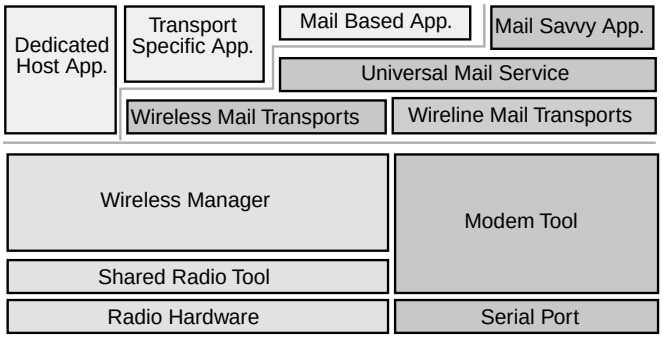


Figure 1: Marco Communications Architecture

The radio communications section of the architecture includes the radio hardware, Shared Radio Tool and the Wireless Manager. Together, these components provide all of the software required for wireless communication on a DataTAC network. The Shared Radio Tool is completely responsible for managing the radio hardware and provides two access points. The first access point is via an endpoint used by the Wireless Manager, which is responsible for providing the Wireless API to clients as well as any user interfaces that deal directly with the radio (such as radio preferences). More detail on these components is given later in the article.

The mail section of the architecture provides the software necessary to support multiple mail transports, bundled mail transport engines, mailed based/savvy applications and a mail editor application. The diagram contains a single component labeled "Mail Transports," which is actually multiple mail transport implementations, all of which share a common structure. In the initial release both Radio Mail and ARDIS Personal Messaging are provided as Mail Transports. The relationship between the Universal Mail Service and individual transport services is detailed later in the article.

The last section includes applications and services other than mail that are provided by third parties. Three different APIs are provided to developers and the choice of API will typically be based on the type of communications service required.

Applications that use a mail based service are layered on top of the Universal Mail Service (UMS). Such applications are shown as “Mail Based" and “Mail Savvy” applications in the diagram. (Mail Based apps are specifically designed to take advantage of the I/O box, Mail Savvy apps are those that provide mail access as an option to communicate data. Examples of Mail Savvy apps are the built-in NotePad and Calendar applications). The UMS API is completely compatible with the existing Mail API in the Newton, while providing additional mechanisms to get information about currently available transports and/or register for receiving reply mail. Applications using this sort of interface are the most generic since they will work across multiple mail transports. In addition the host that is supplying the service and/or data needs only be accessible through mail. The primary drawback to this sort of application is that the store and forward paradigm of mail can result in significant delays.

Some mail transports also provide value added network services (Radio Mail's XMFS, for example). Applications that take advantage of these services are "Transport Specific" applications. Since the exact nature and API of the network service will vary between transports, these applications will only run over the transport for which they were designed. It is difficult to discuss the potential benefits or drawbacks to these types of applications since the value added network service itself will vary considerably across transports.

The last type of application is one that talks directly to the Wireless Manager ("Dedicated Host" applications). These applications use the Wireless API to send and received data across the wireless network. Applications of this type must communicate with a specific host on the wireless network (ARDIS in the US). Since the process for getting access of this type is potentially very expensive and time consuming, this type of application is most likely to be used in vertical markets. Since mail transports use the Wireless API, they can be considered as a special kind of "Dedicated Host" application.

Overview of the Communications Architecture

The radio communications portion of the architecture is shown in more detail in figure 2. The major components are the radio hardware, the Shared Radio Tool, the Wireless Manager, and the Dedicated Radio Tool. As mentioned before, these components work together to provide radio communications over a DataTAC network (such as ARDIS).

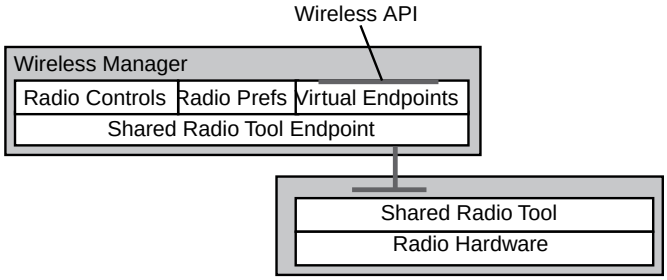


Figure 2: Overview Communications Architecture

The radio provides the physical communications layer and the channel access portion of the data link layer. The Shared Radio Tool (SRT) is a communications tool that provides two separate interfaces. In the current operating system, each communications tool may only be used by a single client at any given time. To accommodate multiple radio clients, the wireless manager has assumed the responsibility for multiplexing clients requests to the radio. As a result, the Shared Radio Tool's normal interface has a data format significantly different than a typical communications tool.

SRT is a dual mode protocol engine (MDC and RDLAP) suitable for use on MFR and SFR DataTAC networks. It implements the portions of the data link layer above channel access, the network layer, and connection management functions.

The Wireless Manager uses an endpoint to communicate with the Shared Radio Tool. All access to the radio from NewtonScript is made through this endpoint. Internally, the Wireless Manager provides components for setting the radio preferences and controlling the radio (on, off, etc.). Most importantly, the Wireless Manager also implements virtual endpoints which provide the Wireless API for clients to use. Multiple clients may instantiate virtual endpoints simultaneously - the Wireless Manager coordinates use of the real endpoint among the multiple virtual endpoints.

Detailed Mail Architecture

The Mail Architecture is shown in detail in Figure 3. This document describes each of the components briefly; more detailed information may be found in Universal Mail Service. Three major components are involved in the sending of mail: an application, the Universal Mail Service, and a mail transport. The application sending the mail may be the Mail Application itself, a third party application layered on top of mail services (mail based applications), or any Newton application that supports the standard mail routing service. The Universal Mail Service provides a consistent API for all mail transports. Applications need not know the difference between the individual mail transports supplied by third parties.

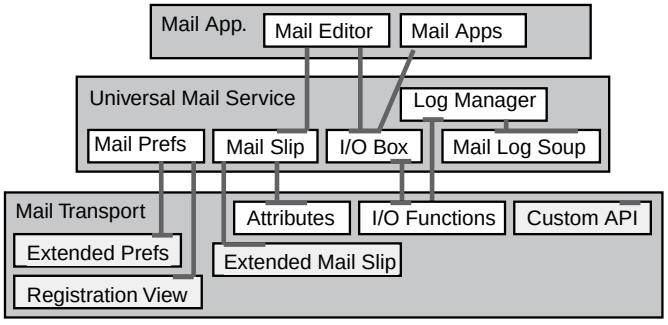


Figure 3: Detailed Mail Architecture

The Mail Application consists of a mail editor and a log overview. The Mail Editor makes use of the Mail Slip and I/O Box provided by the Universal Mail Service. Other applications that need to send or receive mail would also use these interfaces. Universal Mail Service provides an extensible logging capability, the results of which are stored in a Mail Log Soup. The Log Overview provides the user with convenient access to the data in this soup.

The user selects and configures mail transports through the Mail Prefs component. Custom preferences may be optionally supplied by the transport as "Extended Prefs". In addition, the transport may optionally supply a registration view that allows the user to register that transport.

The Mail Slip allows the user to address and submit a piece of mail. It uses the transports' Attributes to determine what features are available for the specific transport. In addition, custom features can be supported by an Extended Mail Slip optionally supplied by the transport.

The Universal Mail Service owns the Mail InBox/OutBox category and makes calls to the transport's I/O Functions to send mail. When receiving mail, the transport should deliver it to the I/O box so that the Universal Mail Service can dispatch it appropriately.

Logging of mail is supported by the Log Manager and the Mail Log Soup. The Log Manager provides convenient functions to the transport to support logging. The log is maintained as a soup whose format will be published.

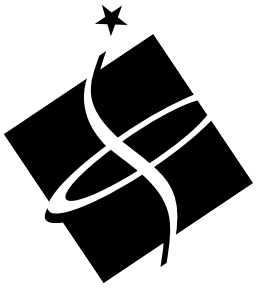
Lastly, a mail transport may optionally provide a Custom API for "Transport Specific" clients. The custom API is often a standard used to provide access to several programming platforms (DOS, Windows, Macintosh, Newton).

For more information about Marco Communicator development opportunities, developers may contact Pat Pahl, Business Development Manager at Motorola, at the following internet address: Patrick_Pahl@msmail.wes.mot.com.

Marco Wireless Communicator, Motorola, InfoTAC and DataTAC are all registered trademarks of Motorola, Inc. Newton, Newton Technology, Newton Script, MessagePad, Newton Toolkit (NTK) and NewtonMail are all registered trademarks of Apple Computer, Inc. ARDIS, ARDIS MG, ARDIS Personal Messaging, RadioMail, SprintNet, CompuServe, America Online, and MCI Mail are all trademarks of their respective companies. NTJ

Newton®

If you have an idea
for an article you'd like to write for Newton
Technology Journal, send it via internet to:
piesysop@applelink.apple.com
or AppleLink: PIESYSOP



STARCORE™

Dear Newton Developer,

We'd like to introduce you to StarCore, the software publishing and distribution arm of the Personal Interactive Electronics Division at Apple Computer, Inc. As a Newton developer, you are already involved in creating products for this exciting technology. There are many ways in which we can build relationships that will benefit you and the Newton platform.

At StarCore, we are actively recruiting titles for the Newton. StarCore can provide developers with a broad range of services and opportunities. The developer creates the software, StarCore provides the packaging, manuals, testing, user studies, marketing and end-user support.

We are anxious to talk with developers about products or concepts they would like to see published or distributed. We are particularly interested in business-oriented applications that would appeal to a mobile professional. We are also looking for products that have connectivity to Macintosh and Windows desktop applications.

Please contact us at:

StarCore
Apple Computer, Inc.
5 Infinite Loop, MS 305-3C
Cupertino, CA 95014
Attn: StarCore